



# National Textile University

## Department of Computer Science

Subject:

Operating System

---

Submitted to:

Sir Nasir

---

Submitted by:

Eman Babar

---

Reg. number:

23-NTU-CS-FL-1148

---

Semester:

5<sup>th</sup>- A

---

# LAB-09

## TASK-1-Binary Semaphore

a)

Code:

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4  #include <unistd.h>
5  sem_t mutex; // Binary semaphore
6  int counter = 0;
7  void* thread_function(void* arg) {
8      int id = *(int*)arg;
9      for (int i = 0; i < 5; i++) {
10         printf("Thread %d: Waiting...\n", id);
11         sem_wait(&mutex); // Acquire
12         // Critical section
13         counter++;
14         printf("Thread %d: In critical section | Counter = %d\n", id,
15                counter);
16         sleep(1);
17         sem_post(&mutex); // Release
18         sleep(1);
19     }
20     return NULL;
21 }
22 int main() {
23     sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
24     pthread_t t1, t2;
25     int id1 = 1, id2 = 2;
26     pthread_create(&t1, NULL, thread_function, &id1);
27     pthread_create(&t2, NULL, thread_function, &id2);
28     pthread_join(t1, NULL);
29     pthread_join(t2, NULL);
30     printf("Final Counter Value: %d\n", counter);
31     sem_destroy(&mutex);
32     return 0;
33 }
```

Output:

The screenshot shows the VS Code interface running on a WSL Ubuntu 24.04 host. The terminal window displays the execution of a C program named Task1\_lab9\_1148.c. The program uses two threads to increment a counter. Thread 1 prints messages indicating it is waiting or in a critical section. Thread 2 also prints similar messages. The output shows both threads interacting with a shared resource (the counter) simultaneously, with Thread 2's increments being visible even while Thread 1 is in a critical section. The final counter value is printed as 10.

```
● ohai1_arwar@SALMAN:~/OS-1148/LAB_8_1148$ gcc Task1_lab9_1148.c -o Task1.out -lpthread
● ohai1_arwar@SALMAN:~/OS-1148/LAB_8_1148$ ./Task1.out
Thread 1: Waiting...
Thread 2: In critical section | Counter = 1
Thread 2: Waiting...
Thread 1: Waiting...
Thread 2: In critical section | Counter = 2
Thread 1: Waiting...
Thread 2: In critical section | Counter = 3
Thread 1: Waiting...
Thread 2: In critical section | Counter = 4
Thread 1: Waiting...
Thread 2: In critical section | Counter = 5
Thread 1: Waiting...
Thread 2: In critical section | Counter = 6
Thread 1: Waiting...
Thread 2: In critical section | Counter = 7
Thread 2: Waiting...
Thread 2: In critical section | Counter = 8
Thread 1: Waiting...
Thread 1: In critical section | Counter = 9
Thread 2: Waiting...
Thread 2: In critical section | Counter = 10
Final Counter Value: 10
● ohai1_arwar@SALMAN:~/OS-1148/LAB_8_1148$
```

b)

## Code:

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <semaphore.h>
4  #include <unistd.h>
5  sem_t mutex; // Binary semaphore
6  int counter = 0;
7  void* thread_function(void* arg) {
8      int id = *(int*)arg;
9      for (int i = 0; i < 5; i++) {
10         printf("Thread %d: Waiting...\n", id);
11         sem_wait(&mutex); // Acquire
12     // Critical section
13     counter++;
14     printf("Thread %d: In critical section | Counter = %d\n", id,
15           counter);
16     sleep(1);
17     sem_post(&mutex); // Release
18     sleep(1);
19     }
20     return NULL;
21   }
22   int main() {
23     sem_init(&mutex, 0, 0); // Binary semaphore initialized to 1
24     pthread_t t1, t2;
25     int id1 = 1, id2 = 2;
26     pthread_create(&t1, NULL, thread_function, &id1);
27     pthread_create(&t2, NULL, thread_function, &id2);
28     pthread_join(t1, NULL);
29     pthread_join(t2, NULL);
30     printf("Final Counter Value: %d\n", counter);
31     sem_destroy(&mutex);
32     return 0;
33 }
```

```
#include <stdio.h>
#include <unistd.h>
sem_t mutex; // Binary semaphore
int counter = 0;
void* thread_function(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 5; i++) {
        printf("Thread %d: Waiting...\n", id);
        sem_wait(&mutex); // Acquire
        // Critical section
        counter++;
        printf("Thread %d: In critical section | Counter = %d\n", id, counter);
        sleep(1);
        // sem_post(&mutex); // Release
        sleep(1);
    }
    return NULL;
}
int main() {
    sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
    pthread_t t1, t2;
    int id1 = 1, id2 = 2;
    pthread_create(&t1, NULL, thread_function, &id1);
    pthread_create(&t2, NULL, thread_function, &id2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Final Counter Value: %d\n", counter);
    sem_destroy(&mutex);
    return 0;
}
```

Thread 2: Waiting...  
Thread 2: In critical section | Counter = 6  
Thread 1: Waiting...  
Thread 1: In critical section | Counter = 7  
Thread 2: Waiting...  
Thread 2: In critical section | Counter = 8  
Thread 1: Waiting...  
Thread 1: In critical section | Counter = 9  
Thread 2: Waiting...  
Thread 2: In critical section | Counter = 10  
Final Counter Value: 10

c)

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex; // Binary semaphore
int counter = 0;
void* thread_function(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 5; i++) {
        printf("Thread %d: Waiting...\n", id);
        sem_wait(&mutex); // Acquire
        // Critical section
        counter++;
        printf("Thread %d: In critical section | Counter = %d\n", id, counter);
        sleep(1);
        // sem_post(&mutex); // Release
        sleep(1);
    }
    return NULL;
}
int main() {
    sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
    pthread_t t1, t2;
    int id1 = 1, id2 = 2;
    pthread_create(&t1, NULL, thread_function, &id1);
    pthread_create(&t2, NULL, thread_function, &id2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Final Counter Value: %d\n", counter);
    sem_destroy(&mutex);
    return 0;
}
```

d)

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t mutex;
sem_t counter;

void *thread_function(void *arg) {
    int i = 0;
    while (i < 10) {
        sem_wait(&mutex);
        if (sem_getvalue(&counter) == 0) {
            printf("Thread 1: In critical section | Counter = %d\n", i);
            sem_post(&counter);
        }
        sem_post(&mutex);
        i++;
    }
    printf("Final Counter Value: %d\n", i);
}

int main() {
    pthread_t thread1, thread2;
    sem_init(&mutex, 1, 1);
    sem_init(&counter, 1, 0);

    pthread_create(&thread1, NULL, thread_function, NULL);
    pthread_create(&thread2, NULL, thread_function, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    sem_destroy(&mutex);
    sem_destroy(&counter);
}
```

### **Comments:**

In this code, semaphore ensures that only one thread enters the critical section at a time while incrementing the shared counter. Then, two threads wait and then update counter safely, print their status and release semaphore. After both threads finish, the final counter value is printed. When we initialize semaphore to 0 then it will block both threads forever so they never enter critical section and as a result, program never ends. And, if we comment “sem\_post(&mutex);), the semaphore is never released, so only the first thread enters once and then both threads block forever. And, if we comment “sem\_wait(&mutex);), both threads enter the critical section at the same time which may cause race conditions.

## **TASK-2-Counting Semaphore**

a)

### **Code:**

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4 #include <unistd.h>
5 sem_t resource_semaphore;
6 void* thread_function(void* arg) {
7     int thread_id = *(int*)arg;
8     printf("Thread %d: Waiting for resource...\n", thread_id);
9     sem_wait(&resource_semaphore); // Wait: decrement counter
10    printf("Thread %d: Acquired resource!\n", thread_id);
11    sleep(2); // Use resource
12    printf("Thread %d: Releasing resource...\n", thread_id);
13    sem_post(&resource_semaphore); // Signal: increment counter
14    return NULL;
15 }
16 int main() {
17     sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads
18     pthread_t threads[5];
19     int ids[5];
20     for (int i = 0; i < 5; i++) {
21         ids[i] = i;
22         pthread_create(&threads[i], NULL, thread_function, &ids[i]);
23     }
24     for (int i = 0; i < 5; i++) {
25         pthread_join(threads[i], NULL);
26     }
27     sem_destroy(&resource_semaphore);
28     return 0;
29 }

```

## Output:

```

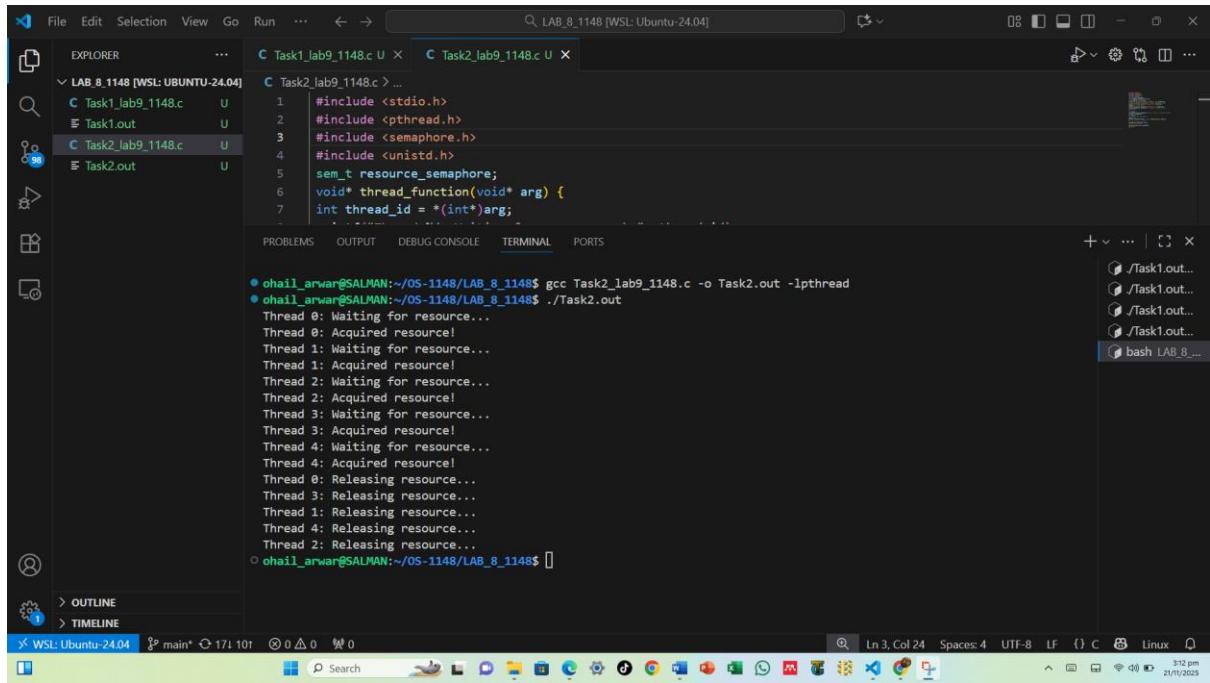
● ohail_arwar@SALMAN:~/OS-1148/LAB_8_1148$ gcc Task2_lab9_1148.c -o Task2.out -lpthread
● ohail_arwar@SALMAN:~/OS-1148/LAB_8_1148$ ./Task2.out
Thread 0: Waiting for resource...
Thread 0: Acquired resource!
Thread 2: Waiting for resource...
Thread 2: Acquired resource!
Thread 1: Waiting for resource...
Thread 1: Acquired resource!
Thread 3: Waiting for resource...
Thread 4: Waiting for resource...
Thread 2: Releasing resource...
Thread 0: Releasing resource...
Thread 4: Acquired resource!
Thread 1: Releasing resource...
Thread 3: Acquired resource!
Thread 4: Releasing resource...
Thread 3: Releasing resource...

```

b)

## Code:

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4 #include <unistd.h>
5 sem_t resource_semaphore;
6 void* thread_function(void* arg) {
7     int thread_id = *(int*)arg;
8     printf("Thread %d: Waiting for resource...\n", thread_id);
9     //sem_wait(&resource_semaphore); // Wait: decrement counter
10    printf("Thread %d: Acquired resource!\n", thread_id);
11    sleep(2); // Use resource
12    printf("Thread %d: Releasing resource...\n", thread_id);
13    sem_post(&resource_semaphore); // Signal: increment counter
14    return NULL;
15 }
16 int main() {
17     sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads
18     pthread_t threads[5];
19     int ids[5];
20     for (int i = 0; i < 5; i++) {
21         ids[i] = i;
22         pthread_create(&threads[i], NULL, thread_function, &ids[i]);
23     }
24     for (int i = 0; i < 5; i++) {
25         pthread_join(threads[i], NULL);
26     }
27     sem_destroy(&resource_semaphore);
28     return 0;
29 }
```



c)

## Code:

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4 #include <unistd.h>
5 sem_t resource_semaphore;
6 void* thread_function(void* arg) {
7     int thread_id = *(int*)arg;
8     printf("Thread %d: Waiting for resource...\n", thread_id);
9     sem_wait(&resource_semaphore); // Wait: decrement counter
10    printf("Thread %d: Acquired resource!\n", thread_id);
11    sleep(2); // Use resource
12    printf("Thread %d: Releasing resource...\n", thread_id);
13    //sem_post(&resource_semaphore); // Signal: increment counter
14    return NULL;
15 }
16 int main() {
17     sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads
18     pthread_t threads[5];
19     int ids[5];
20     for (int i = 0; i < 5; i++) {
21         ids[i] = i;
22         pthread_create(&threads[i], NULL, thread_function, &ids[i]);
23     }
24     for (int i = 0; i < 5; i++) {
25         pthread_join(threads[i], NULL);
26     }
27     sem_destroy(&resource_semaphore);
28     return 0;
29 }
```

## Output:

The screenshot shows the Visual Studio Code interface running in a WSL Ubuntu 24.04 terminal window. The code editor displays two files: Task1\_lab9\_1148.c and Task2\_lab9\_1148.c. The Task2\_lab9\_1148.c file is open and shows the provided C code for a semaphore-based thread pool. The terminal below the editor shows the execution of the program, outputting messages from each thread as it waits for, acquires, and releases the shared resource semaphore. The file explorer on the left shows the project structure with files Task1.out and Task2.out.

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <semaphore.h>
4 sem_t resource_semaphore;
5 void* thread_function(void* arg) {
6     int thread_id = *(int*)arg;
7     printf("Thread %d: Waiting for resource...\n", thread_id);
8     sem_wait(&resource_semaphore); // Wait: decrement counter
9     printf("Thread %d: Acquired resource!\n", thread_id);
10    sleep(2); // Use resource
11    printf("Thread %d: Releasing resource...\n", thread_id);
12    //sem_post(&resource_semaphore); // Signal: increment counter
13    return NULL;
14 }
15 int main() {
16     sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads
17     pthread_t threads[5];
18     int ids[5];
19     for (int i = 0; i < 5; i++) {
20         ids[i] = i;
21         pthread_create(&threads[i], NULL, thread_function, &ids[i]);
22     }
23     for (int i = 0; i < 5; i++) {
24         pthread_join(threads[i], NULL);
25     }
26     sem_destroy(&resource_semaphore);
27     return 0;
28 }
```

```
ohail_arwar@SALMAN:~/OS-1148/LAB_8_1148$ gcc Task2_lab9_1148.c -o Task2.out -lpthread
ohail_arwar@SALMAN:~/OS-1148/LAB_8_1148$ ./Task2.out
Thread 0: Waiting for resource...
Thread 0: Acquired resource!
Thread 1: Waiting for resource...
Thread 1: Acquired resource!
Thread 2: Waiting for resource...
Thread 2: Acquired resource!
Thread 3: Waiting for resource...
Thread 4: Waiting for resource...
Thread 1: Releasing resource...
Thread 2: Releasing resource...
Thread 0: Releasing resource...
[]
```

d)

## Code:

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4 #include <unistd.h>
5 sem_t mutex; // Binary semaphore
6 int counter = 0;
7
8 //FUNCTION FOR THREAD
9 void* thread_function1(void* arg) {
10     int id = *(int*)arg;
11     for (int i = 0; i < 5; i++) {
12         printf("Thread %d: Waiting...\n", id);
13         sem_wait(&mutex); // Acquire
14         // Critical section
15         counter++;
16         printf("Thread %d: In critical section | Counter = %d\n", id,
17                counter);
18         sleep(1);
19         sem_post(&mutex); // Release
20         sleep(1);
21     }
22     return NULL;
23 }
24
25 // FUNCTION FOR THREAD2
26
27 void* thread_function2(void* arg) {
28     int id = *(int*)arg;
29     for (int i = 0; i < 5; i++) {
30         printf("Thread %d: Waiting...\n", id);
31         sem_wait(&mutex); // Acquire
32         // Critical section
33         counter--;
34         printf("Thread %d: In critical section | Counter = %d\n", id,
35                counter);
36         sleep(1);
37         sem_post(&mutex); // Release
38         sleep(1);
39     }
40     return NULL;
41 }
42 int main() {
43     sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1
44     pthread_t t1, t2;
45     int id1 = 1, id2 = 2;
46     pthread_create(&t1, NULL, thread_function1, &id1);
47     pthread_create(&t2, NULL, thread_function2, &id2);
48     pthread_join(t1, NULL);
49     pthread_join(t2, NULL);
50     printf("Final Counter Value: %d\n", counter);
51     sem_destroy(&mutex);
52     return 0;
53 }
54
```

## Output:

```

#include <stdio.h>
#include <pthread.h>

Thread 1: Waiting...
Thread 2: Waiting...
Thread 2: In critical section | Counter = -1
Thread 1: In critical section | Counter = 0
Thread 2: Waiting...
Thread 2: In critical section | Counter = -1
Thread 1: Waiting...
Thread 1: In critical section | Counter = 0
Thread 2: Waiting...
Thread 2: In critical section | Counter = -1
Thread 1: Waiting...
Thread 1: In critical section | Counter = 0
Thread 2: Waiting...
Thread 2: In critical section | Counter = -1
Thread 1: Waiting...
Thread 1: In critical section | Counter = 0
Final Counter Value: 0

```

### Comments:

This program uses a counting semaphore which is initialized to 3, and allow only 3 threads to access the shared resource at the same time, while the remaining 2 threads wait until a slot becomes free. If we comment `sem_wait(&resource_semaphore)`, all five threads enter the resource section together with no limit, so the semaphore provides no control. If we comment `sem_post(&resource_semaphore)`, threads will never release the resource and cause only first three threads to run while remaining two stays blocked forever. With one added function, both threads safely increments or decrements the shared counter using a binary semaphore so only one thread enters the critical section at a time.

## TASK-3-Comparison of Mutex lock and Binary Semaphore

### **Similarities:**

- Both are used for protecting shared resources in multithreading.
- Both allow only one thread to access a critical section at a time.
- Both help in preventing race conditions and data consistency.

### **Differences:**

Features	<b>Mutex Lock</b>	<b>Binary Semaphore</b>
Ownership	Only the thread that locks it can unlock it.	there is no ownership, any thread can signal (post) it.
Purpose	Designed only for mutual exclusion i.e., locking.	Designed for both mutual exclusion and signalling between threads.
Blocking	If already locked, thread waits until released.	If value is 0 then waits, but if value is 1 then it enters immediately.
Speed	Slightly faster	Slightly slower
Usage Restriction	Always used in pairs i.e., lock and unlock.	It is used with wait ( <code>sem_wait</code> ) and post ( <code>sem_post</code> ).